Views and Indexes

Instructor: David Martin

Reference:
A First Course in Database Systems,
3rd edition, Chapter 2.3 and 8.1-8.4
(except 8.2.3 and 8.4.3)

SQL Language

- Data Definition Language (DDL)
  - Modify structure of data
  - CREATE, DROP, ALTER
- Data Manipulation Language (DML)
  - Access and modify data
  - SELECT, INSERT, DELETE, UPDATE
- Data Control Language (DCL)
  - Control access to the data (security)
  - GRANT, REVOKE
- Databases also have Utilities, such as Backup/Restore
  - Syntax not specified in the SQL standard

VIEWS: Motivation for Views

- Views help with logical data independence, allowing you to retrieve data as if it matched the description in the view.

  CREATE VIEW < view-name> AS < view-definition> ;

  CREATE VIEW ParamountMovies AS
  SELECT title, year
  FROM Movies
  WHERE studioName = ‘Paramount ' ;

- You may now ask queries on ParamountMovies as if it were a table:
  SELECT title FROM ParamountMovies WHERE year=1976;
  - Composition in SQL is powerful: Tables, Queries, Views

More Views

Movies (title , year , length , genre , studioName , producerC #)

MovieExec (name , address , cert # , netWorth)

```
CREATE VIEW MovieProd AS
    SELECT title, name, genre
    FROM Movies , MovieExec
    WHERE producerC # = cert # ;

    SELECT DISTINCT genre
    FROM MovieProd
    WHERE name = 'George Lucas';
```

Renaming Attributes in CREATE VIEW

Movies (title , year , length , genre , studioName , producerC #)

MovieExec (name , address , cert # , netWorth)

```
CREATE VIEW MovieProd(movie_title, prod_name, movie_genre) AS
    SELECT title, name, genre
    FROM Movies , MovieExec
     WHERE producerC# = cert# ;

   SELECT DISTINCT movie_genre
    FROM MovieProd
    WHERE prod_name = 'George Lucas';
```

What is a View?

- A view can include any SQL SELECT statement
  - Including UNION, Aggregates, GROUP BY, HAVING, ORDER BY, etc.
- A view is <u>not</u> stored as a table
  - The tables underlying the view are stored in the database, but only the description of the view is in the database
  - … although some systems support MATERIALIZED VIEWS
- But view can be used in many (not all) of the same ways as tables
  - Views can be queried
  - Views can be defined on views, as well as on tables!

Queries on Views and Tables

```
CREATE VIEW ParamountMovies AS
       SELECT title , year
       FROM Movies
       WHERE studioName = ’Paramount ' ;

SELECT DISTINCT starName
FROM ParamountMovies , StarsIn
WHERE title = movieTitle AND year = movieYear ;

CREATE VIEW ParamountStars AS
      SELECT DISTINCT starName
      FROM ParamountMovies , StarsIn
      WHERE title = movieTitle AND year = movieYear ;
```

DROP VIEW

```
CREATE VIEW ParamountMovies AS
    SELECT title , year
    FROM Movies
    WHERE studioName = ’Paramount ' ;

DROP View ParamountMovies;
```

- What happens if you execute the following?
  - SELECT * FROM ParamountMovies;
  - SELECT * FROM Movies;

View Updates

- Modifications (INSERT/DELETE/UPDATE) are allowed on certain views with simple definitions
  - SELECT query involving a single relation R
  - R not involved in a subquery
  - Attributes of R not appearing in the SELECT clause are assignable with NULL or a default
- Modifications on more complex views are disallowed because:
  - Constraint on underlying table would be violated, or
  - View update is not well-defined
- The SQL rules are complex

- Example for Movies(title, year, length, genre, studioName , producerC#)

```
CREATE VIEW ParamountMovies AS
    SELECT title , year
    FROM Movies
    WHERE studioName = 'Paramount ' ;

INSERT INTO ParamountMovies VALUES ('StarTrek', 1979);
```

- OK if the other columns of Movies (besides title and year) have defaults or allow nulls

Types of Relations in an RDB

- Three types of relations
  1. Base relations (stored relations, tables)
     - These are tables that contain tuples and can be modified or queried.
  2. Views (derived relations, virtual relations)
     - Views are relations that are defined in terms of other relations but they are not stored. They are constructed only when needed.
  3. Temporary tables
     - These are tables (representing intermediate results) that are constructed by the query execution engine during the processing of a query and discarded when done.

INDEXES: Motivation for Indexes

- Searching an entire table may take a long time:

```
SELECT *
FROM Movies
WHERE studioName = ‘Disney’ AND year = 1990;
```

If there were 100 Million movies, searching them might take a while. An index (e.g., a B-Tree) would allow faster access to matching movies.

If a table is updated, all relevant Indexes on that table are immediately <u>automatically</u> modified within the same transaction.

Primary Index

SELECT *
FROM Movies
WHERE Title = Monsters, Inc.
AND Year = 1990

MovieIndex

Title	Year	Ptr
Alien	1979	5
Back to the Future	1985	6
Jurassic Park	1998	3
Life Is Beautiful	1997	8
Monsters, Inc.	1990	2
Pretty Woman	1990	1
Princess Mononoke	1997	7
Star Wars IV	1977	4

Binary search

Primary key

Movies

Title	Year	Length	Genre	Studio	…
Princess Mononoke	1997	134	Fantasy	DENTSU	…
Monsters Inc.	1990	121	Animation	Dreamworks	…
Jurassic Park	1998	145	Adventure	Disney	…
Star Wars IV	1977	121	Sci-fi	LucasFilm	…
Alien	1979	117	Sci-fi	20th Century Fox	
Back to the Future	1985	116	Sci-fi	Universal	
Pretty Woman	1990	119	Romantic	Disney	…
Life Is Beautiful	1997	116	Comedy	Melampo	

Data Structures and Algorithms

- Computer science offers a variety of data structures and algorithms to support indexing
  - E.g., binary search trees, B-trees, B+trees and associated algorithms
  - Note: a great deal of research and progress fundamental computer science has been driven by database requirements

```
SELECT *
FROM Movies
WHERE Year = 1990
      AND Studio = Disney
```

Secondary Indexes (1)

YearIndex

Year	Ptr
1977	4
1979	5
1985	6
1990	2
1990	1
1997	8
1997	7
1998	3

Primary key

Movies

Title	Year	Length	Genre	Studio	…
Princess Mononoke	1997	134	Fantasy	DENTSU	…
Monsters Inc.	1990	121	Animation	Dreamworks	…
Jurassic Park	1998	145	Adventure	Disney	…
Star Wars IV	1977	121	Sci-fi	LucasFilm	…
Alien	1979	117	Sci-fi	20th Century Fox	
Back to the Future	1985	116	Sci-fi	Universal	
Pretty Woman	1990	119	Romantic	Disney	…
Life Is Beautiful	1997	116	Comedy	Melampo	

Ptr	Studio
5	20th Century Fox
1	DENTSU
3	Disney
7	Disney
2	DreamWorks
4	LucasFilm
8	Melampo
6	Universal

```
SELECT *
FROM Movies
WHERE Year = 1990
      AND Studio = Disney
```

Secondary Indexes (2)

YearIndex

Year	Ptr
1977	4
1979	5
1985	6
1990	2
1990	1
1997	8
1997	7
1998	3

Primary key

Movies

Title	Year	Length	Genre	Studio	…
Princess Mononoke	1997	134	Fantasy	DENTSU	…
Monsters Inc.	1990	121	Animation	Dreamworks	…
Jurassic Park	1998	145	Adventure	Disney	…
Star Wars IV	1977	121	Sci-fi	LucasFilm	…
Alien	1979	117	Sci-fi	20th Century Fox	
Back to the Future	1985	116	Sci-fi	Universal	
Pretty Woman	1990	119	Romantic	Disney	…
Life Is Beautiful	1997	116	Comedy	Melampo	

Ptr	Studio
5	20th Century Fox
1	DENTSU
3	Disney
7	Disney
2	DreamWorks
4	LucasFilm
8	Melampo
6	Universal

CREATE INDEX

SELECT *
FROM Movies
WHERE studioName = ‘Disney’ AND year = 1990;

How much would each of these indexes help?

CREATE INDEX YearIndex ON Movies(year);

CREATE INDEX StudioIndex ON Movies(studioName);

CREATE INDEX SYIndex ON Movies(studioName,year);

How much would each of the indexes help if the WHERE clause was just
year = 1990?

Disadvantages of Indexes?

- Why not put indexes on every attribute, or even on every combination of attributes that you might query on?
  - Huge number of indexes
  - Space for indexes
  - Cache impact of searching indexes
  - Update time for indexes when table is modified

Selection of Indexes

- Selecting indexes involves evaluation of a trade-off:
  - Great speed up for the affected queries, vs.
  - Significant performance penalties

- Common choices for indexing
  - Primary key
    - Used frequently
    - At most one (data) *page* will be retrieved
  - Attribute that's "almost a key"
    - Relatively few tuples for a given value for that attribute
  - Attribute for which values are frequently specified in queries
  - Attribute for which tuples are "clustered"
    - Common values for the attribute are stored nearby (on a small number of pages)
- Depends mostly on what queries are used heavily

Index Utilization

- SQL statements don't specify use of indexes, so they don't have to be modified when you change what's indexed!
  - Database Optimizer tries to figure out "the best"/"a good" way to execute each SQL query.
  - All the tuples in a Relation can be scanned directly, without using indexes, so indexes aren't necessary … except for performance.
  - Some systems have ways that you can tell the Optimizer what to do. This has advantages and disadvantages. (What are they?)

- Many SQL systems (including PostgreSQL) have an EXPLAIN PLAN statement, so that you can see what plan the optimizer chooses for a SQL statement.